RISC-V Business

A Computer Design Implementing the RISC-V Intruction Set Architecture

Cesar Avalos, Jacob Fustos, Stephen Longofono

May 2018

Contents

List of Figures

1 Introduction & Overview

This document details the design and implementation of a computer system containing a RISC-V core running on the Nexys 4 FPGA target. Our goal was to start with an ISA, design a core that implemented that ISA, link that core to the outside world by designing memory-mapped peripheral controllers, and then create software that could run on that core. We chose RISC-V for our ISA as it was open source and, even though it has a minimal instruction set, it is powerful enough to run a full operating system. We chose the Nexys 4 FPGA because it would be able to meet the requirements of our design, and we had experience doing past projects with it. We chose BusyBox for our operating system as it runs on the Linux Kernel that had already been ported to RISC-V and was minimal enough to run on our system. Once these 3 tasks were complete we would then need to integrate them all together.

The RISC-V ISA is an open-source standard for research and development of computer architectures maintained by the RISC-V Foundation. In general, the ISA follows the philosophy of previous RISC implementations, favoring simplicity and regularity over specialization. RISC-V offers constructs to support all major macro and micro-architectures, and as such is a worthwhile study to complete our undergraduate education. RISC-V is modularized into a base integer set of instructions and operations; these fundamental instructions make up the minimum viable product for any RISC-V implementation[2]. In order to implement the kernel, we would need to include a collection of extensions to the base set, dubbed "RV64G". The RV64G extension composes the minimum instruction set for use in a standard Linux system. It includes the base

set of integer arithmetic instructions "I", the multiplication and division extension "M", the atomic instruction extension "A", the single-precision floating-point extension "F", and the double precision floating-point extension "D".

The Nexys 4 trainer board includes a serial ROM device that we use to store our application and system code. There was no previously designed controller for this device so we implemented our own using the timing diagram from the chip's datasheet. The controller was then memory mapped through our bus unit so that the core could write and read from it. The same was done with the board's DDR RAM and UART, but controllers were already available and just needed to be integrated into the system bus. The 16 LEDs on the board were also memory mapped so the the program could write to them for debugging. The system bus is controlled by the MMU which also implements SV39 address translation as defined by the RISC-V specification. This is important as the LINUX kernel requires address translation to manage memory.

For the software we began by setting up a development environment that is maintained by the RISC-V foundation. Once that was done we needed something to run code on while the core was being developed. We chose the spike simulator as it fully implemented the extensions that we needed and was considered the "gold standard" for instruction execution. This was important to us as it helped us to clarify parts of the ISA that were confusing or not well defined. A few simple programs were developed to run on the core in a bare metal fashion so that we could test the core along the way. A script was developed to work as a terminal to communicate with the device as we designed and implemented a debugger unit that could also communicate over the UART using a serial protocol that we designed.

2 Project Objectives

The objectives of our project are listed below. Note that they have been modified from our original plan due to time constraints and unforeseen difficulties with integration.

- Develop a simple serial core to support arithmetic instructions, basic branching, loads, and stores. These constitute the "32I", "64I", and "M" extensions of the RISC-V ISA
- Develop support for privileged instructions and interrupts, to allow an operating system to run on our core
- Develop a Debug unit that could communicate over UART
- Develop a MMU to do address translation
- Develop or integrate controllers for RAM, ROM, UART, and LEDs
- Develop a bus unit that maps those controllers to physical memory addresses
- Develop a Bootloader to transfer code from the ROM to RAM
- Modify the Berkeley Boot Loader to work with our system
- Create a system image that could be run on our system

3 Problem Constraints & Project Viability

Our largest constraint was time. We did not have enough man hours to have some of the features that a modern day computer would have because we either did not have time to implement them, or their inclusion would cause other units to become too complex. Our original design included several of the features below, as described in the design decisions section. However, as time progressed, it became clear that we needed to narrow our focus. For this reason, the following items were removed:

- Atomic instructions
- Pipelined core

- Out of Order core
- Translation Look Aside Buffer
- Level 1 or Level 2 caches
- High-speed data buses
- VGA controller
- Keyboard controller
- Additional MMU translation schemes
- USB flash drive controller
- Full Linux distribution build
- Ethernet controller
- Linux networking support

The clocking crystal on the Nexys 4 board is only 100 MHz, and primarily because of the lack of caching and the low speed buses each instruction can take up to 300 clock cycles to complete. This restricts the core to operate at about 300 K instructions per second. While this is not a remarkable speed, simple programs are still able to run on the core, and the goal of the project was not to design a core that could compete with other modern day processors. Further improvement to the project could have raised the instruction frequency to above 1 MHz.

With respect to financial viability, our project is inexpensive by design. Since we implemented on an FPGA, and we intended to support common commercial, off-the-shelf peripherals, the only expenses we incurred are labor and two FPGAs for development.

We selected the Nexys4 with DDR memory from Digilent as our platform, priced at $320.00 each. This platform includes USB, RS32, and micro-USB interfaces as well as a micro-SD card reader to serve as persistent storage. The two boards are the only hardware we required.

Our labor varied. While school was in session, we had 5 hours in class and an additional 5 hours outside of class each week. For our three members, that put us at 30 hours per week for the 16 week semester. Over the winter break, we were able to devote as much as 20 hours each, for a total of 60 hours in each of the 5 weeks of the winter break. Using a local rate of $25.00 hourly for a software engineering intern, we incurred $(30 \cdot 16 + 60 \cdot 5) \cdot \$25.00 = \$19,500.00$ in labor cost.

4 Major Components & Descriptions

This section serves as an overview of our design and its individual components. In many ways, our biggest trade-off concern was developer time and experience. This project was as much about implementing a RISC-V computer as it was about learning how the disparate parts we have studied come together. As such, we needed to balance our ambitions with what we had the ability to learn and implement in a year's time. To address this, we simplified and modularized our designs such that they were relatively easy to implement and test. Needless to say, this came at the expense of overall performance. The plan to develop a working, simple design and iterate on it ensured a working design to demonstrate at the end.

4.1 The Core

The main core follows the lead of the base integer core as presented in the RISC-V textbook [1]. Figure 1 depicts the high-level flow of instructions and data through the core. The control unit maintains control lines for all other modules, to dictate what action should be taken at any given time (including halting). The control unit and the MMU work together to queue, execute, and write back instructions and their results.

When an instruction is ready, the decoder identifies its type and breaks out the addresses, function and opcodes, and immediate values. The register file coordinates reading and writing to the standard set of

64-bit general purpose registers. The ALU handles operations on the data loaded through the register file, and feeds the results back to the MMU or the register file.

The implementation of the floating point unit and the atomic unit was not deemed of utmost importance for the revised scope of the project. While both units are hard requirements for supporting a Linux kernel, the floating point unit could be handled in software, while the absence of the atomic unit would prove to be more problematic. Therefore the revised scope of the project foregoes the Linux kernel in favour of less complex programs. Functionally, the FPU and the ATU would sit in parallel to the ALU, with signals to both the control unit and the MMU to handle synchronization.

At the control level, the control unit and the control status registers track the privilege mode and handle privileged instructions. This is closely related to the handling of interrupts and traps, so the control status registers and the control unit are one and the same with the interrupt handler module. This internal state machine is capable of halting the pipeline and interacting with the MMU to handle interrupts as outlined in the RISC-V privileged instructions specifications [4].

Finally, to facilitate development and troubleshooting, the core has asynchronous buses with the contents of the register and control status registers fed to the debugger unit. This gave us a play-by-play recap of what went through the core after we had moved it out of simulation. As you probably can tell, the core is the coolest part of the project by far.

4.2 The Debugger

The debugger has three main components. The debugger uses UART to interface with the PC, as such the first and second components are the UART RX and TX components (receive and transmit respectively). The third component is the debug controller, which handles the debugging logic. The debugger unit receives the registers and control status from the core, and the debugger sends a pause signal to the core. The idea behind the debugger is very simple, the user sends a command, the RX component receives said command, the debugger logic identifies said command and prepares the action to be performed, the action is performed, and finally the TX responds back with the outcome. It should be noted that some complications might arise if the processor is pipelined along the way, and the design must be modified accordingly.

4.3 MMU

The MMU sits between the core and the system bus. It takes in virtual addresses from the core and uses SV39 paging standards from the RISC-V ISA to convert them into physical addresses to place on the bus. If errors occur during translation, the MMU returns the proper exception code back to the core.

4.4 System Bus

The system bus sits between the MMU and the peripherals. It begins by taking in a physical address from the MMU and decides what peripheral device that address corresponds to. If no devices correspond to that address, an error signal is returned. If the device does exist the bus accesses the control unit for that device in the correct way. The System bus unit also contains a system timer that can be read from and written to by accessing the correct address. There is also a timer compare register that can be set and is used to trip the timing interrupt in the core. It also contains a special address that can be written to to trip the machine mode software interrupt in the core. It also controls the LEDs on the board and maps them to a physical address.

4.5 ROM

Running code is a major aspect of the project. Initially the programs were trivial, few lines of code. As such, hard-coding programs into the core was a practical solution, programs were not often changed. As the project matured, more complex programs were introduced. A point was reached were it was no longer practical to hard-code programs in the core. The FPGA board has a ROM chip (S25FL128) which proved to be the solution to this problem. The user simply flashes the instructions to the ROM chip via the memory configuration options offered by Vivado. No controller was available to utilize the ROM chip in this manner, a new controller had to be written from the ground-up using the component's data sheet [3].

4.6 RAM, UART

The main feature of the Nexys-4 DDR FGPA board is its 128 MiB DDR2 Memory. Both the RAM and UART had already available controllers that were used for the project

5 Software

5.1 Spike

The RISC-V Simulator(Spike) was studied and understood as a first step. This served two purposes: it is a functional core, so we were able to learn much about RISC-V implementations that support the full 64G extensions. Also, it served as a functional unit for development while the board was in the early stages. Because the simulator's interfaced lacked necessary features for debugging, such as the ability to view the control registers, Spike was usually run through gdb. A script was then written for gdb that gives the user the impression that they are simply using spike with additional features.

5.2 The Berkeley Bootloader

The Berkeley bootloader contains all machine code for other software such as the Linux kernel. We were able to do a special build of it that implemented all of the 'F' and 'D' instructions in software. This allowed us to remove the floating point unit from our design drastically reducing our design time. Since these instructions now cause an invalid instruction trap and execution of a handler, this causes programs that use floating point instructions to execute much more slowly.

5.3 Linux Kernel, BusyBox

Both of these programs were already ported to RISC-V; we only needed to compile them with the particular settings that were appropriate. Since spike takes these elf files and pre-loads them into memory, we dumped Spike's memory and used that as a flat binary that we could copy into RAM for execution.

5.4 Debugger Terminal

Both the core and the debugger unit utilize the same UART port. To simplify its operation the debugger unit has a special protocol that it uses to send out register information. Unfortunately, this meant we needed additional support to be able to use our debugger with a UART terminal. A script was written that allows characters from a normal program to flow through to the terminal, and a magic number is used to switch to processing information from the debugging unit. Once all of the information from the debugger is dumped, the terminal switches back to normal mode.

5.5 Binary to VHDL

Early on, we were running binaries out of internal ROM arrays within the FPGA. Those arrays were hand coded with instruction data. For programs the were thousands of bytes long, we developed a script that could convert a flat binary into a list of comma-separated bitstrings compatible with VHDL. This allowed us to write any program, and copy the corresponding bitstrings directly into the instantiation of the array.

5.6 Boot Loader

A boot loader was programmed that is placed at the beginning of ROM. When the system first loads up, this is the program that is running. It takes care of copying the system image from ROM into RAM, and then using a CRC calculator it checks that the RAM area's CRC matches a pre-calculated CRC. Once that is done, the first instruction of the system binary is jumped into.

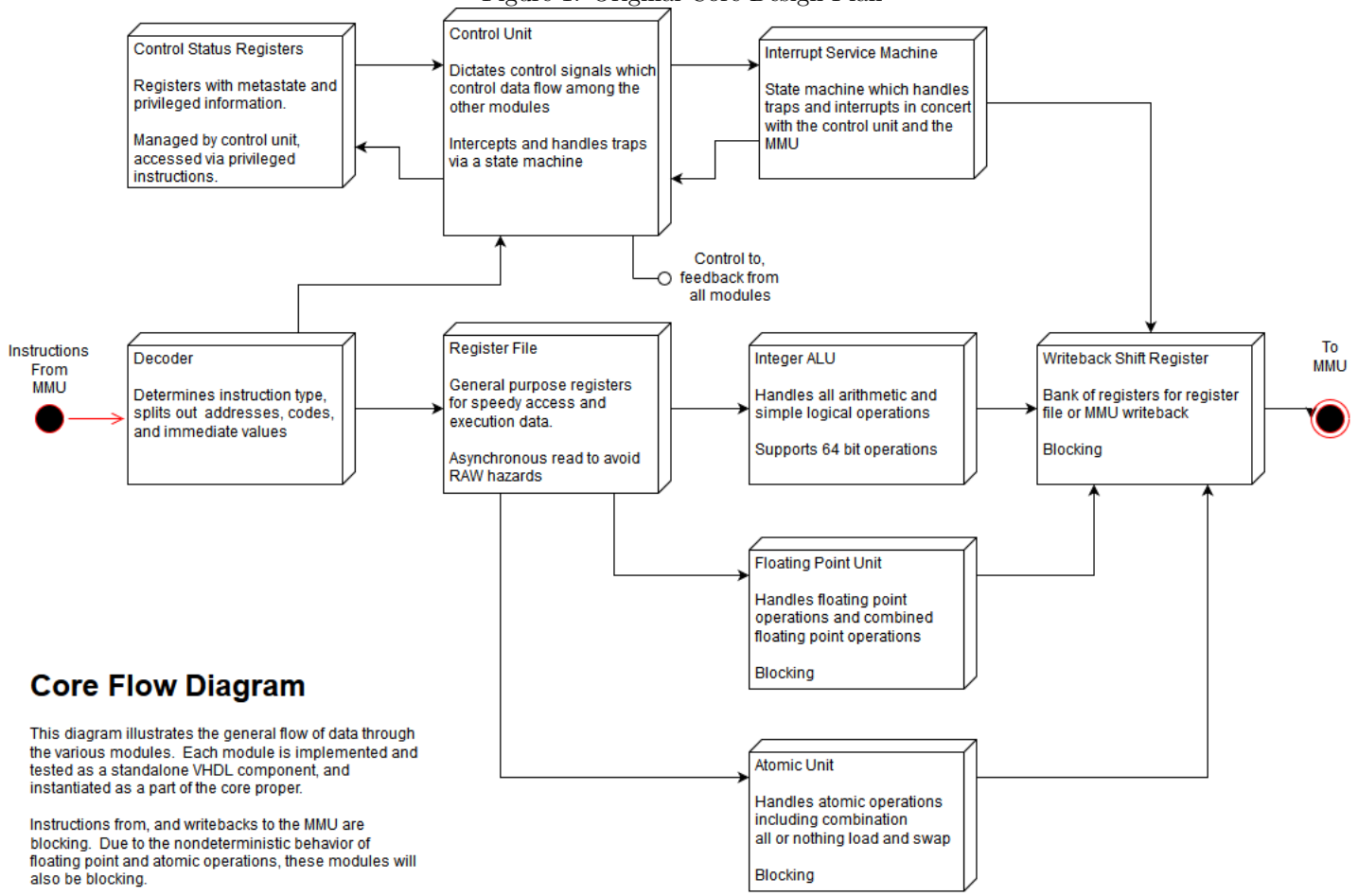
5.7 Miscellaneous Bare Metal Programs

Two simple programs were created that could run on our system without the use of an operating system. The first was the echo server that simply prints out a title and then waits for a character to be sent in on the UART, and then echoes that character back out over UART. The LEDs on the board show a count of the total characters transferred in this way. The second program was the Fibonacci server. This program reads in an input value for the index, calculates the Fibonacci sequence at that in index and then prints the solution to the console.

6 Design Documents

Here, we have included our design documents for the system, including our original goal design, flow charts for various execution streams in the core, and our final block diagram.

Figure 1: Original Core Design Plan



Core Flow Diagram

This diagram illustrates the general flow of data through the various modules. Each module is implemented and tested as a standalone VHDL component, and instantiated as a part of the core proper.

Instructions from, and writebacks to the MMU are blocking. Due to the nondeterministic behavior of floating point and atomic operations, these modules will also be blocking.

The control unit dictates the data flow via active signalling to each other module. This allows all modules to be halted while blocking operations complete. In general, the MMU and control unit will interact and share via the control status registers.

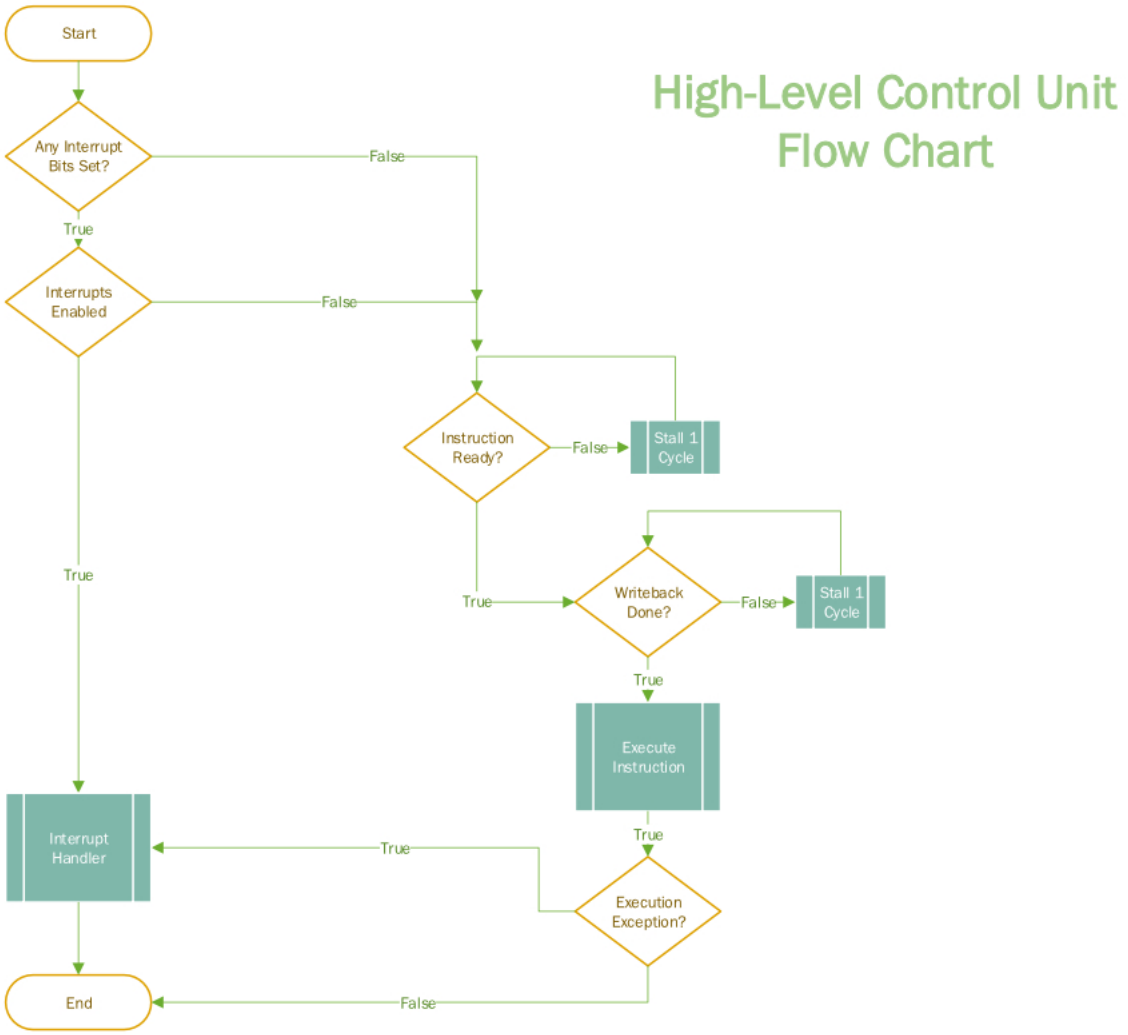


Figure 2: The Control Unit. This depicts the logical flow chart for the operation of the Control Unit

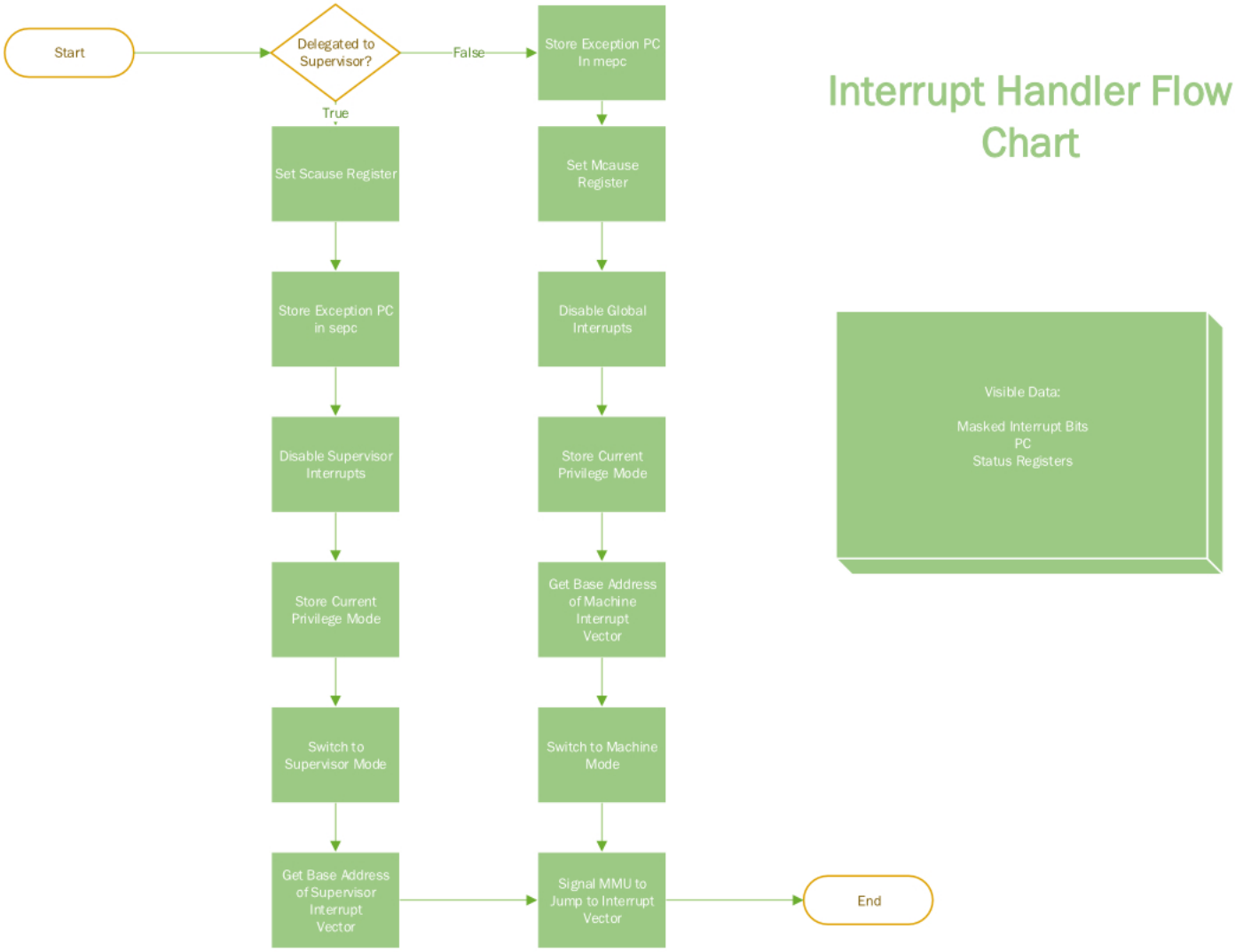


Figure 3: Interrupt Handling. This depicts the logical flow chart for handling interrupts and exceptions.

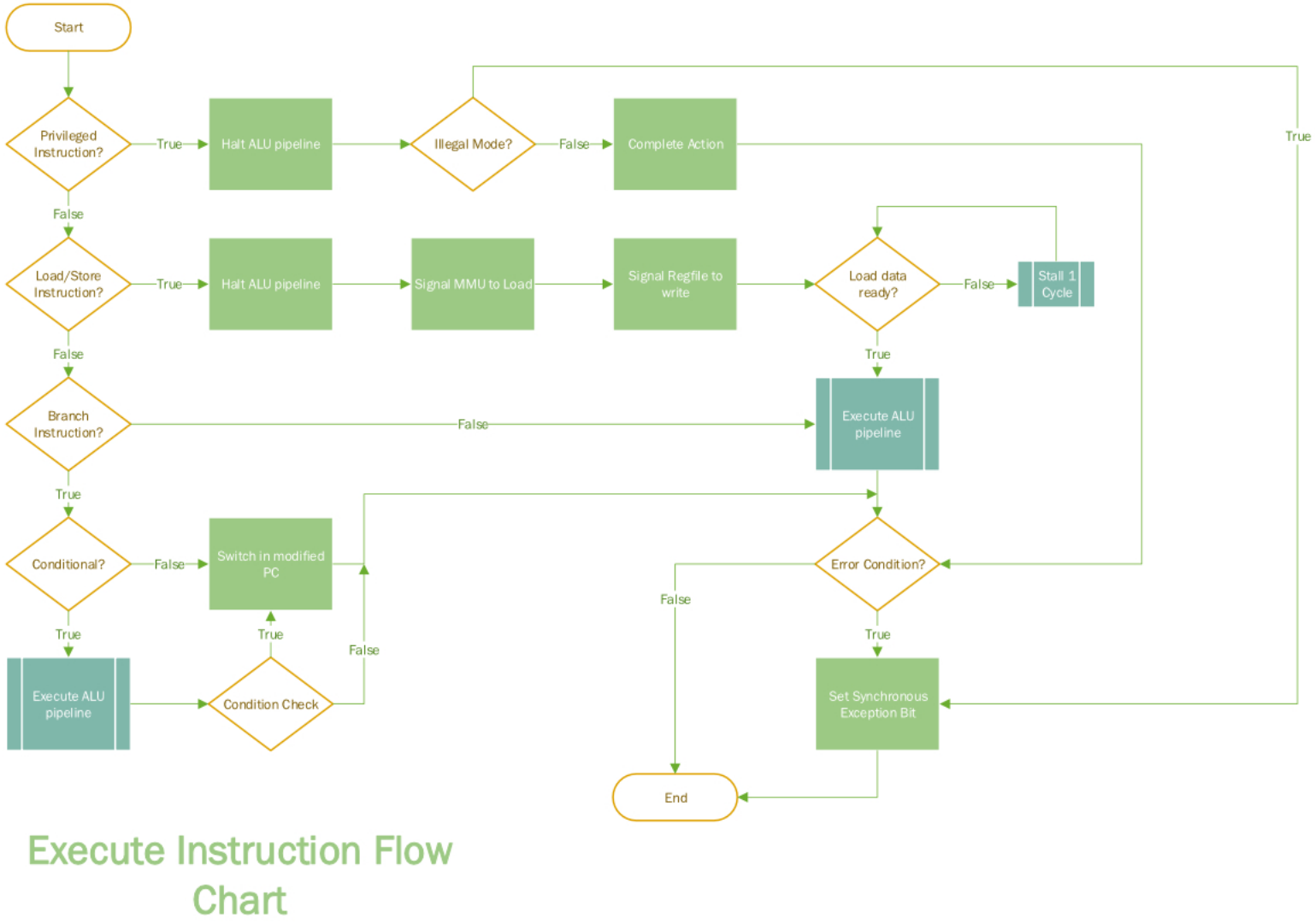


Figure 4: Instruction Execution. This depicts the logical flow chart for instruction execution under normal conditions.

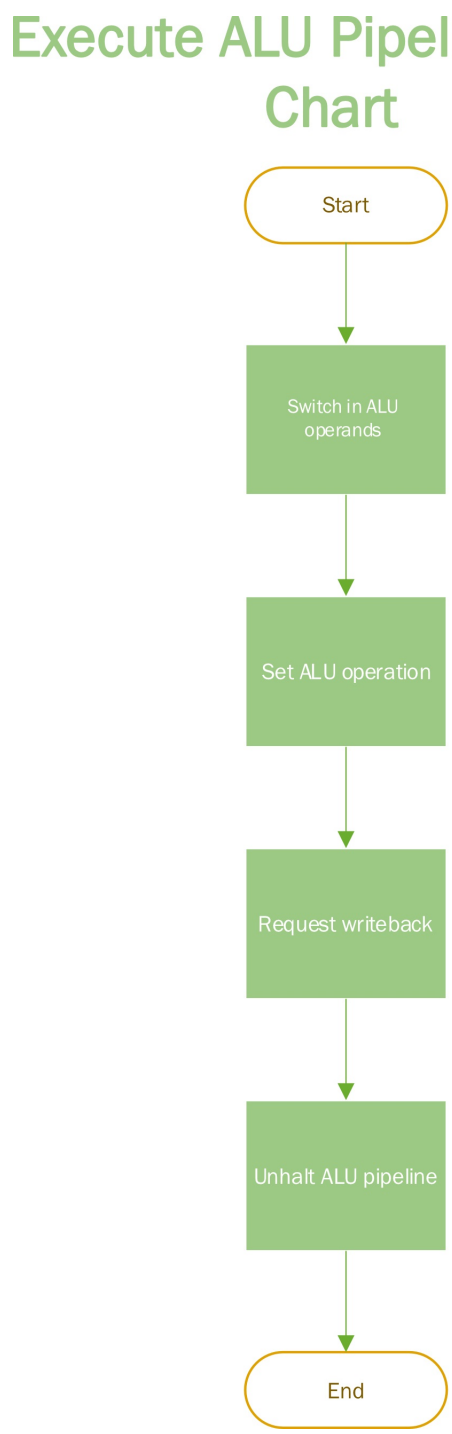


Figure 5: ALU Operations. This depicts the logical flow chart for the execution of instructions handled by the ALU.

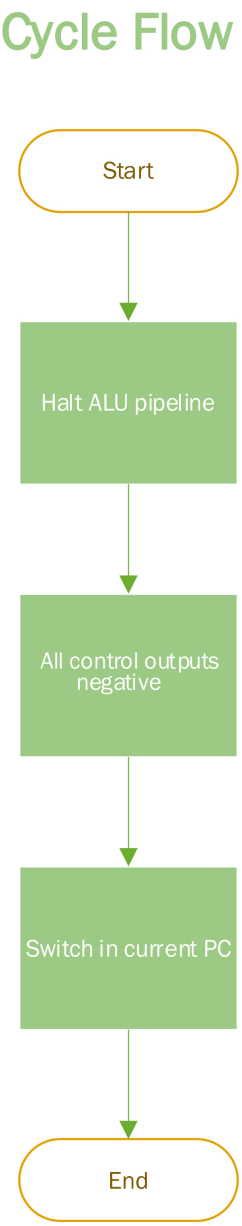


Figure 6: Stall Cycles. This depicts the logical flow chart for how the core handles stalls or other halts of normal execution.

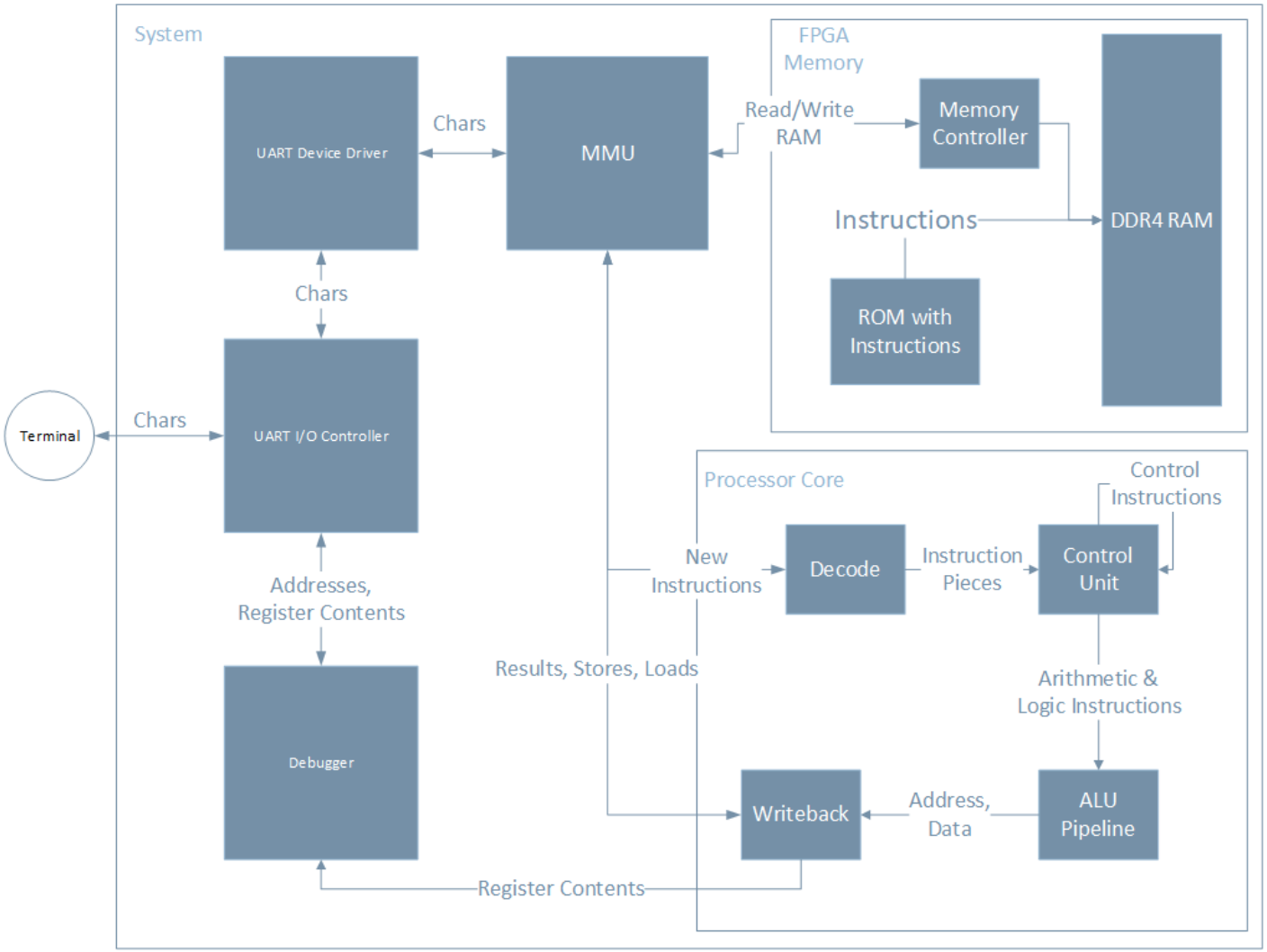


Figure 7: Final Block Diagram. This depicts the system design as of the time of this writing.

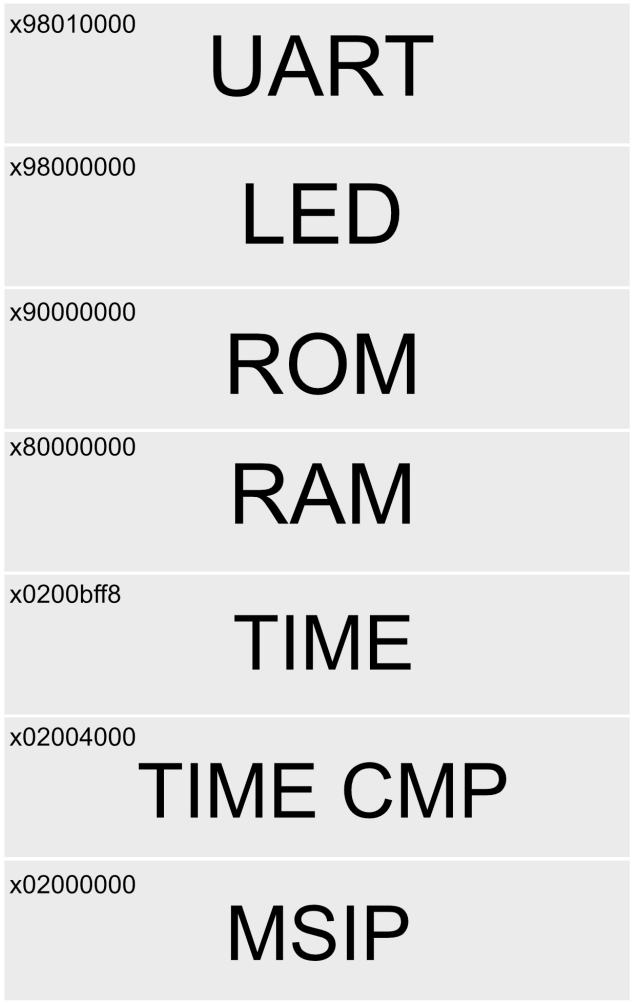


Figure 8: Memory Map.

7 Standards & Protocols in Use

- RISC-V User-level ISA Specification, version 2.2—We wrote our processor code to implement the ISA per these specifications.
- RISC-V Privileged ISA Specification Draft, version 1.10—We wrote our processor code to adhere to the privilege mode conventions described in these specifications.
- VHDL IEEE std 1076-1993—The Vivado IDE only fully supports the 1993 standard for VHDL. All our system components and processor code made use of this standard.
- DDR Memory Specification—The DDR specification guided our work on interacting with the DDR2 memory on the Nexys4.
- AXI4 Protocol—This protocol is the standardized interface to various IP modules used in the MMU.
- UART Serial Protocol—This protocol is used by our debugger unit to communicate register contents, and by the system to take inputs from a keyboard and produce output for the terminal.
- Linux Kernel version 4.12.0—We used this Linux kernel as a starting point to build our binary for the system.
- Busybox version 1.26.2—We used this software as our terminal and interface to the operating system.

8 Design Requirements & Constraints

8.1 Design

The ISA is the list of requirements for our implementation. Tools provided by the RISC-V foundation have been helpful along the way, but the ISA is the definitive document. The behavior of the instructions, CSR registers, and register file are all dictated by the ISA document.

8.2 Memory

Our memory constraints include the amount of available DDR2 RAM and the size of the FPGA fabric. As it stands, our implementation uses 48% of the available FPGA fabric. We also encountered problems with the latency of our memory reads with respect to the clock speed of our processor. Even though the memory is clocked faster than our processor, it takes on average 300 ns to access 2 bytes. Since we are running at 100 MHz, we cannot read fast enough to handle our 8 byte instructions. The bottleneck of our processor is reading from RAM.

8.3 Costs

By design, our project is meant to be open-source and affordable. We do not have a specific limit for price of materials. We need only an existing computer to act as a thin client over UART, and a NEXYS4 DDR FPGA. In that sense, so long as we do not purchase anything else for our project, we have already met our cost constraints.

8.4 Time

The development timeline is the primary driver of our design decisions and how thoroughly we implement the ISA. Our development time is limited to two semesters, further still by the responsibilities of our other classes and employment. By regularly revisiting the scope of the project against our planned schedule, we are able to adapt our implementation so it is finished on time.

8.5 Complexity

Due to the time constraints mentioned above, we need to tightly scope what is implemented. Our experience with computer architecture and system design is limited, and designing a processor is a monumental task. We have taken steps to improve our chances of success, setting aside optimizations until we have a working implementation we are satisfied with. One such adaptation is the use of a behavioral VHDL style, trading efficiency and speed of design for ease of implementation.

8.6 Real-time

Our system should ideally run as fast a possible, that is, as close to the clock speed of the FPGA as possible. As with costs, we do not have a specific constraint on how fast our system should run. That said, our self-imposed requirements are to run with the best CPI possible. Much of the core and MMU design is such that an instruction can complete in as few cycles as possible. For the most part, this means we traded space complexity for time complexity. For example, allowing Vivado to synthesize our multipliers with DSP slices uses more of the FPGA fabric, allowing instructions to complete in lockstep with the system clock.

9 Design Decisions & Alternate Designs Considered

There are innumerable ways we could have designed our computer—ultimately, we adopted the familiar architecture approach presented in our computer architecture coursework. The Von Neumann architecture presented by Hennessey and Patterson is widely employed in RISC-V implementations, and its modular nature makes it easy to extend and improve individual pieces as need be. Note that this includes decisions about our intended design; some of these decisions were reversed for reasons described in the discussion section.

9.1 Processor Core

The core was designed to follow the modular, Von Neumann architecture presented in Computer Architecture: A Quantitative Approach by David Patterson and John Hennessey. A control unit handles the bulk of the operating logic, sending signals to control the inputs and outputs among internal components and to the external modules like the MMU and the peripherals. Within the core, each logical operation is delegated to its own module; a decoder processes binary instructions, an ALU handles arithmetic instructions, a register file stores and serves data from a bank of registers. The advantages of this approach are that each module can be tested in isolation, and each module improved upon or otherwise modified in isolation. The trade-off is the complexity of implementing an interface among the modules and keeping the modules in lockstep with the core clock.

Initially, we also considered a monolithic, state-machine style design. The latter would have made atomic instructions, interrupts, and general flow of information relatively simple. Each possible behavior could be captured in a non-deterministic finite automaton, and reduced to the minimal equivalent deterministic automaton. Such a design would eliminate timing issues altogether, and by nature eliminate unexpected or undefined behavior at clock boundaries. However, the rigid design means that even small changes could require the state machine be redesigned and optimized. We decided against his approach because it would limit the extensibility of our design and complicate the addition of pipelining in the future.

9.2 System Design

Initially, we had planned to implement the 32-bit core with extensions for multiplication, 64-bit instructions, floating-point operations, privileged instructions, and atomic operations. These "standard" extensions are compatible with existing implementations that will run Linux. As the semester progressed, it became clear that this was too ambitious. We did not have enough time to complete and test all of the above and still have a working product to demonstrate. Instead, we decided to implement incremental versions of the core, each adding in some core functionality and stubbing out or omitting the rest. The first version implemented only arithmetic instructions, the second control and privileged instructions, the third load and jump instructions,

and the final version atomic instructions. Since it was possible to implement the operating system without floating point operations, we opted to omit it completely.

9.3 System Bus

We considered implementing multiple paging schemes in the MMU as that is what was done in the simulator, and it would not have been too difficult to do. Then we discovered the paging mode used by the Linux Kernel, we decided to implement that instead. We wanted to implement a VGA and keyboard driver so that the computer could interact with the world. We eventually settled on just using UART as the input and output to the console. This allowed us to only design a single driver and still allows the system to interact through a serial connection with another computer. We also considered having a USB drive being the hard disc for our system. We decided against this as we did not have enough USB slots, and we would have needed to add additional drivers to the system. Instead the file system is read only and is stored on the ROM chip.

9.4 Operating System and ABI

Originally we considered a full Linux distribution for embedded projects. This was abandoned for a bare bones model for the added complexity and time it would have taken to run the programs and compile. We wanted to created system tests to ensure the proper operation of the different instructions under corner case conditions. This was abandoned for time reasons, and because we felt as though the debugger unit would be able to help us troubleshoot any problem instructions.

10 Issues from Design to End-of-Life

10.1 Continued Development & Design Maintenance

Our prototype is very much a work in progress, and as such a user can expect regular updates and upgrades through the end of its lifetime. At a minimum, we expect to develop support for floating point operations, vectored interrupts, and updates to keep our system compatible with the Linux kernel. The size and scope of our project is such that we expect maintenance to be a part-time job.

10.2 Technical Support

Our prototype design will exist as an open source project on Github, maintained and administered by the original developers. Beyond issuing upgrades and bug fixes through version control, we will accept technical feedback in the form of Github Issues. The built-in issue tracking will allow us to track and resolve issues identified internally and externally. Should we extend our design to other FPGA boards, we will also provide constraints and if necessary alternate modules to support the new hardware.

10.3 Upgrades

Our prototype design includes most of the features we originally planned, in the simplest form we were able to realize them in. At minimum, we anticipate developing the rest of the features of the original plan. This includes support for floating point operations, vectored interrupts, atomic instructions, full Linux support, and a VGA controller for a terminal. After that, performance improvements and optimizing our design takes precedence; many parts of our core and MMU would benefit from a combinatorial design, and the use of caching and translation look-aside buffers would greatly improve our memory bottlenecks.

Given the modular nature of the ISA, we could also provide support for vectored operations, pipelined execution, out-of-order execution, and compressed instructions. All of the above would require a substantial redesign of the core, but they are included here for completeness' sake.

10.4 Updates & Maintenance

Updates and bug fixes will be released through Github. Besides the upgrades mentioned above, critical changes to maintain compatibility with the Linux kernel will be pushed on an as-needed basis. Over time, we will adopt production versions to delimit major (backwards incompatible) changes to our design.

10.5 End of Life

Our product will exist until either the platform or the ISA becomes irrelevant. Extending our work to be compatible with other FPGAs will remove the former. The latter is unlikely. We expect our product to haunt us for the rest of our days.

11 Ethical Considerations

This section outlines a number of ethical concerns that should be considered before using our work in a production setting. The main concern is that our work is not ready for a production setting. Our processor is very much a prototype, and as such will not be effective under many of the conditions that a production processor might be. This project was an exploration of what we could do with the RISC-V ISA and should be treated as such in any application of our work.

11.1 Assumptions and Testing conditions

Our work closely follows the Von-Neumann architecture for a serial RISC-V processor as laid out in Hennessey and Patterson's Computer Architecture: A Quantitative Approach. We did our testing with the Vivado development suite, using their built-in simulator and CAD tools. The implicit assumption in all our designs is that the system clock is running at the Nexys4 reference clock frequency of 100MHz. To test our work, we built a testbench in Vivado to feed in instructions one after another and write the results back to the register file. We also built a debugger unit which wrote results to a UART terminal to verify our implementation on hardware. Beyond this, we did very little additional testing. As such, our work is not appropriate for any critical application, or any application where correct results matter. We had limited development time, so we abandoned unit testing after the ALU was complete. We want to take this opportunity to apologize to our professors—we know better but we did it anyway.

11.2 Known Defects and Usage Notes

Our system is incomplete, so many of the features that one might expect have yet to be implemented. Our system does not implement atomic or floating point operations, both of which are normally required for running Linux. Our system bus and MMU have some limitations due to the way the RAM and ROM are interfaced on the Nexys4; every instruction read from and written back to RAM and ROM must be completed in 4 smaller parts. This effectively cuts the performance of our processor by 75%.

11.3 End User concerns

The end user is expected to have sufficient technical knowledge to understand an develop computer architecture at the RTL level. All of our documentation is directed at users of this level. In many cases, we have completely omitted documentation and comments, in the style of Linux system programmers. It stands to reason that our work is not appropriate for use by the general public, but we state it explicitly here in case anyone tries to use our work. This work is provided as open source under the MIT license.

11.4 Discussion

We encountered a number of obstacles along the way that forced us to pivot our development plans. First, we had a hard time judging how long each portion of the product would take to complete. Second, we assumed that there would be sufficient documentation in the simulators such that we could infer design choices that

were not explicitly stated in the ISA. Finally, we divided the project in such a way that we were working largely in isolation, which made integration problematic.

As noted in the introduction, we used the project as a learning experience. The RISC-V ISA and its implementations were completely new to us, and we have only cursory experience with the design of a complete modern computer. An unfortunate side effect of this fact is that we were in a poor place to judge how long any one part of our system would take.

On a related note, we were also poorly equipped to judge the completeness of existing implementations. We looked to the SPIKE simulator and some other implementations of processors to discern some of the implementation details not explicitly stated in the ISA. This led unforeseen delays when we encountered some components we needed for our system that did not exist in other implementations. We also realized that parts of the simulation took shortcuts that we could not take in a physical implementation.

The last major challenge was the development approach we took. We worked too independently, and we failed to adapt appropriately to our changing goals. We worked largely independently up until the last month of the project. When it came time to integrate, some of our design choices were incompatible. This was further complicated by the fact that we had fallen behind on a few goals, so we had not done as well as we could have on documentation and incremental testing. Each of us had a good idea of what we had done and the state of our own contribution, but it was difficult to determine what changes needed to happen to make our parts compatible.

Given the benefit of hindsight, we would have chosen more modest goals, and worked more closely as a team on the primary components. There was no way we could have known in advance about all the issues we encountered, but we would have been better prepared to face them with a more integrated approach. Working as a team on the individual components would have allowed us all to be on the same page and made it easier to catch design errors or oversights. This approach also would have allowed us to react to unforeseen delays more appropriately, since we all would have a better idea of exactly how far along we were toward our goals.

In the end, we had to abandon our original modular core design for a state-machine based design. Integration timing issues made it replicate our simulation results in hardware. By using a monolithic state machine at the top level of our design, the various timing and interfacing issues were easier to debug. This also allowed us to make use of most of our work to date, and showcase how they work on more sophisticated programs.

12 Conclusion & Executive Summary

We were able to create a RISC-V core that is implemented on the Nexys 4 board that can execute simple programs. We chose a very ambitious project. Designing the individual components and bringing them together in a cohesive system stretched our ability and took us to the limits of what we understand of computer system design. Along the way, we gained priceless experience with managing a large-scale project and balancing project goals with life. We also had the opportunity to dive into a variety of application-specific topics that we would not have otherwise seen as a part of the curriculum. While we met most of our design goals, we did fail to run Linux on our system and did not achieve full support for the "64G" extension. We learned so much along the way that we could hardly call the project a failure. We would like to take this opportunity to thank our faculty mentor for his excellent guidance, and all of our professors at KU for their support throughout the year. Rock Chalk!

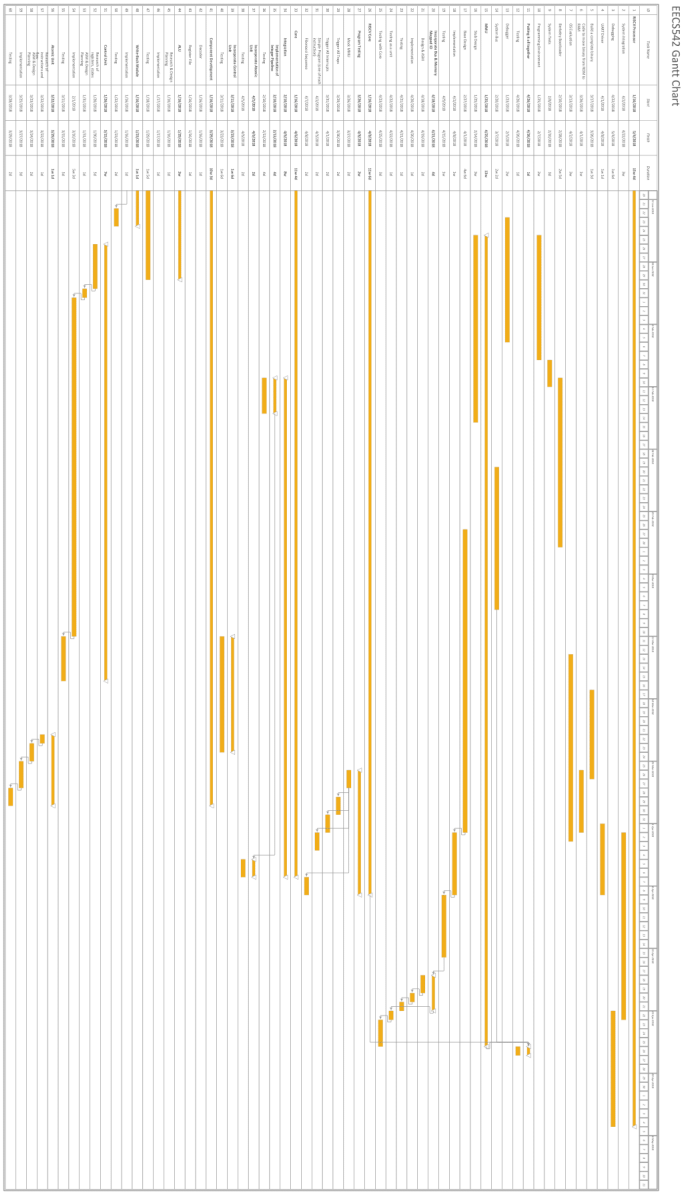
12.1 Final Project Budget

Table 1: Project Budget

Item	Description	Quantity	Per Unit	Total Cost
Labor	Developer time in hours	780	$25.00	$19500.00
Nexys4 DDR FPGA	The FPGA on which our core is implemented	2	$320.00	$640.00
Grand Total		$20140.00		

12.2 Gantt Chart

Figure 9: Project Gantt Chart



References

[1] David Patterson and John Hennessy. *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*. Morgan Kaufmann, Burlington, MA, 2017.

[2] David Patterson and Andrew Waterman. *The RISC-V Reader: An Open Architecture Atlas*. Strawberry Canyon, Berkeley, CA, 2017.

[3] *128 Mbit 3.0V SPI Flash Memory*. S25FL128S. Rev. *O. Cypress. Mar. 2018.

[4] RISC-V Foundation. *The RISC-V Instruction Set Manual Volume II: Privileged Architecture, v1.10*. URL: `https://riscv.org/specifications/privileged-isa/`.